

*Lógica
de negocio
básica*

lección5

Has convertido tu modelo del dominio en una aplicación web plenamente funcional. Esta aplicación ya es bastante útil de por sí, aunque aún puedes hacerle muchas mejoras. Transformemos pues tu aplicación en algo más serio, y de paso, aprendamos algunas cosas interesantes sobre OpenXava.

Empezaremos por añadir algo de lógica de negocio a tus entidades para hacer de tu aplicación algo más que un simple gestor de base de datos.

5.1 Propiedades calculadas

Quizás la lógica de negocio más simple que puedes añadir a tu aplicación es una propiedad calculada. Las propiedades que has usado hasta ahora son persistentes, es decir, cada propiedad se almacena en una columna de una tabla de la base de datos. Una propiedad calculada es una propiedad que no almacena su valor en la base de datos, sino que se calcula cada vez que se accede a la propiedad. Observa la diferencia entre una propiedad persistente y una calculada en el listado 5.1.

Listado 5.1 Diferencia entre una propiedad persistente y una calculada

```
// Propiedad persistente
private int quantity; // Tiene un campo, por tanto es persistente
public int getQuantity() { // Un getter para devolver el valor del campo
    return quantity;
}
public void setQuantity(int quantity) { // Cambia el valor del campo
    this.quantity = quantity;
}

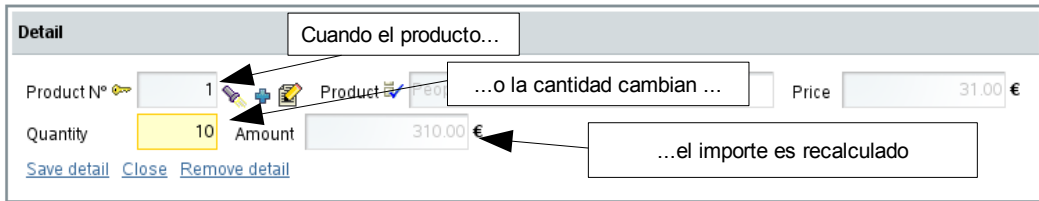
// Propiedad calculada
public int getAmount() { // No tiene campo ni setter, sólo un getter
    return quantity * price; // con un cálculo
}
```

Las propiedades calculadas son reconocidas automáticamente por OpenXava. Puedes usarlas en vistas, listas tabulares o cualquier otra parte de tu código.

Vamos a usar propiedades calculadas para añadir el elemento “económico” a nuestra aplicación Invoicing. Porque, tenemos líneas de detalle, productos, cantidades. Pero, ¿qué pasa con el dinero?

5.1.1 Propiedad calculada simple

El primer paso será añadir una propiedad de importe a Detail. Lo que queremos es que cuando el usuario elija un producto y teclea la cantidad el importe de la línea sea recalculado y mostrado al usuario, tal como muestra la figura 5.1.

**Figura 5.1 Comportamiento de la propiedad calculada amount en Detail**

Añadir esta funcionalidad a tu actual código es prácticamente añadir una propiedad calculada a Detail. Simplemente añade el código del listado 5.2 al código de Detail.

Listado 5.2 Propiedad calculada amount en la clase Detail

```
@Stereotype("MONEY")
@Depends("product.number, quantity") // Cuando el usuario cambie producto o cantidad
public BigDecimal getAmount() {      // esta propiedad se recalculará y se redibujará
    return new BigDecimal(quantity).multiply(product.getPrice());
}
```

Es tan solo poner el cálculo en `getAmount()` y usar `@Depends` para indicar a OpenXava que la propiedad `amount` depende de `product.number` y `quantity`, así cada vez que el usuario cambia alguno de estos valores la propiedad se recalculará.

Ahora has de añadir esta nueva propiedad a la vista de Detail (listado 5.3).

Listado 5.3 Añadir la propiedad amount a la vista de Detail

```
@View(members="product; quantity, amount") // amount añadida
public class Detail extends Identifiable {
```

El único detalle que nos queda es modificar la vista Simple de Product para mostrar el precio. Puedes verlo en el listado 5.4.

Listado 5.4 Añadir la propiedad price a la vista de Product

```
@View(name="Simple", members="number, description, price") // price añadido
public class Product {
```

Nada más. Tan solo necesitas añadir el *getter* y modificar las vistas. Ahora puedes probar los módulos Invoice y Order para ver la propiedad `amount` en acción.

5.1.2 Usar @DefaultValueCalculator

La forma en que calculamos el importe de la línea de detalle no es la mejor. Tiene, al menos, dos inconvenientes. El primero es que el usuario puede querer tener la posibilidad de cambiar el precio unitario. Y segundo, si el precio de un producto cambia los importes de todas las facturas cambian también, y esto no es

bueno.

Para evitar estos inconvenientes lo mejor es almacenar el precio de cada producto en cada línea de detalle. Añadamos pues una propiedad persistente `pricePerUnit` a la entidad `Detail`, y calculemos su valor desde `price` de `Product` usando un `@DefaultValueCalculator`. De tal forma que consigamos el efecto que puedes ver en la figura 5.2.

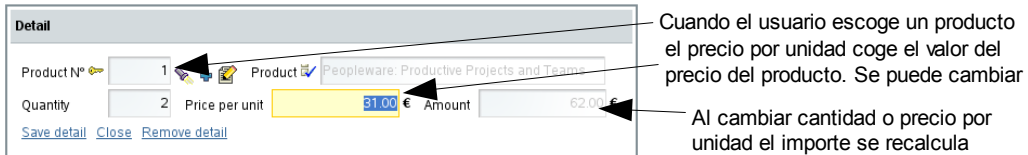


Figura 5.2 Entidad Detail con `pricePerUnit` como propiedad persistente

El primer paso es obviamente añadir la propiedad `pricePerUnit`. Añade el código del listado 5.5 a tu entidad `Detail`.

Listado 5.5 La propiedad `pricePerUnit` de la entidad `Detail`

```
@DefaultValueCalculator(
    value=PricePerUnitCalculator.class, // Esta clase calcula el valor inicial
    properties=@PropertyValue(
        name="productNumber", // La propiedad productNumber del calculador...
        from="product.number") // ... se llena con el valor de product.number de la entidad
)
@Stereotype("MONEY")
private BigDecimal pricePerUnit; // Una propiedad persistente convencional...

public BigDecimal getPricePerUnit() { // ... con sus getter y setter
    return pricePerUnit==null?
        BigDecimal.ZERO:pricePerUnit; // Así nunca devuelve nulo
}

public void setPricePerUnit(BigDecimal pricePerUnit) {
    this.pricePerUnit = pricePerUnit;
}
```

`PricePerUnitCalculator` contiene la lógica para calcular el valor inicial. Simplemente lee el precio del producto. Observa el código de este calculador en el listado 5.6.

Listado 5.6 `PricePerUnitCalculator` calcula el valor por defecto para `pricePerUnit`

```
package org.openxava.invoicing.calculators; // En el paquete calculators

import org.openxava.calculators.*;
import org.openxava.invoicing.model.*;

import static org.openxava.jpa.XPersistence.*; // Para usar getManager()

public class PricePerUnitCalculator implements ICalculator {

    private int productNumber; // En calculate() contendrá el número de producto
```

87 Lección 5: Lógica de negocio básica

```
public Object calculate() throws Exception {
    Product product = getManager() //getManager() de XPersistence
        .find(Product.class, productNumber); // Busca el producto
    return product.getPrice(); ← Returns its price
}

public void setProductNumber(int productNumber) {
    this.productNumber = productNumber;
}

public int getProductNumber() {
    return productNumber;
}
}
```

De esta forma cuando el usuario escoge un producto el campo de precio unitario se rellena con el precio del producto, pero, dado que es una propiedad persistente, el usuario puede cambiar este valor. Y si en el futuro el precio del producto cambiara este precio unitario de la línea de detalle no cambiaría.

Esto implica que has de adaptar la propiedad calculada amount (listado 5.7).

Listado 5.7 La propiedad amount de Detail adaptada para usar pricePerUnit

```
@Stereotype("MONEY")
@Depends("pricePerUnit, quantity") // pricePerUnit en vez de product.number
public BigDecimal getAmount() {
    return new BigDecimal(quantity)
        .multiply(getPricePerUnit()); // getPricePerUnit() en vez de product.getPrice()
}
```

Ahora getAmount() usa productPerUnit como fuente en lugar de product.price.

Finalmente, hemos de modificar la vista de Detail para que muestre la nueva propiedad (listado 5.8).

Listado 5.8 La vista de Product ahora incluye pricePerUnit

```
@View(members="product; quantity, pricePerUnit, amount") // pricePerUnit añadida
public class Detail extends Identifiable {
```

Y la vista Simple de Product para que no muestre el precio (listado 5.9).

Listado 5.9 Vista Simple de Product ahora no incluye price

```
@View(name="Simple", members="number, description") // price quitado
public class Product {
```

Es decir, hemos arreglado la interfaz de usuario para que muestre la propiedad persistente (y modificable por el usuario) pricePerUnit, en vez del precio del producto.

También sería bonito ver estas nuevas propiedades en la colección. Para hacerlo, edita la entidad `CommercialDocument` y modifica la lista de propiedades a mostrar, como muestra el listado 5.10.

Listado 5.10 Añadir `pricePerUnit` y `amount` a `details` en `CommercialDocument`

```
@ListProperties(
    "product.number, product.description, " +
    "quantity, pricePerUnit, amount") // pricePerUnit y amount añadidos
private Collection<Detail> details = new ArrayList<Detail>();
```

Actualiza el esquema de la base de datos, prueba los módulos `Order` e `Invoice` y podrás observar el nuevo comportamiento al añadir líneas de detalle.

5.1.3 Propiedades calculadas dependientes de una colección

También queremos añadir importes a `Order` e `Invoice`. Tener IVA, importe base e importe total es indispensable. Para hacerlo solo necesitas añadir unas pocas propiedades calculadas. La figura 5.3 muestra la interfaz de usuario para estas propiedades.

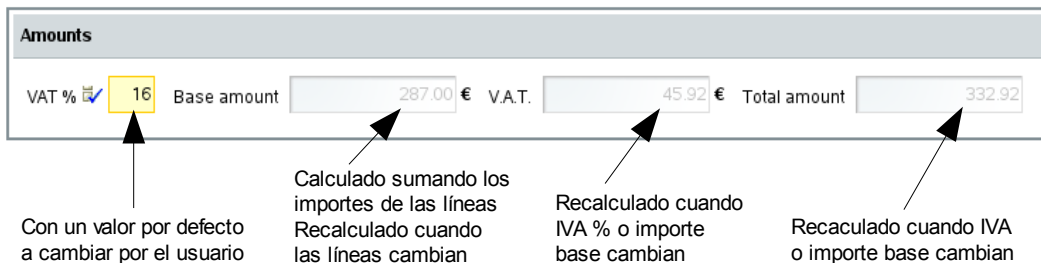


Figura 5.3 Propiedades calculadas en `CommercialDocument`

Empecemos con `baseAmount`. El listado 5.11 muestra su implementación.

Listado 5.11 Propiedad calculada `baseAmount` en `CommercialDocument`

```
@Stereotype("MONEY")
public BigDecimal getBaseAmount() {
    BigDecimal result = new BigDecimal("0.00");
    for (Detail detail: getDetails()) { // Iteramos por todas la líneas de detalle
        result = result.add(detail.getAmount()); // Acumulamos el importe
    }
    return result;
}
```

La implementación es simple, se trata de sumar los importes de todas las líneas.

La siguiente propiedad a añadir es `vatPercentage` que se usará para calcular el IVA. El listado 5.12 te muestra el código para esta propiedad.

Listado 5.12 La propiedad persistente vatPercentage en CommercialDocument

```

@Digits(integerDigits=2, fractionalDigits=0) // Para indicar su tamaño
@Required
private BigDecimal vatPercentage;

public BigDecimal getVatPercentage() {
    return vatPercentage==null?
        BigDecimal.ZERO:vatPercentage; // Así nunca devuelve nulo
}

public void setVatPercentage(BigDecimal vatPercentage) {
    this.vatPercentage = vatPercentage;
}

```

Puedes ver como vatPercentage es una propiedad persistente convencional. En este caso usamos @Digits (una anotación del entorno de validación Hibernate Validator) como una alternativa a @Column para especificar el tamaño.

Continuaremos añadiendo la propiedad vat. La puedes ver en el listado 5.13.

Listado 5.13 La propiedad calculada vat en CommercialDocument

```

@Stereotype("MONEY")
@Depends("vatPercentage") // Cuando vatPercentage cambia vat se recalcula y revisualiza
public BigDecimal getVat() {
    return getBaseAmount() // baseAmount * vatPercentage / 100
        .multiply(getVatPercentage())
        .divide(new BigDecimal("100"));
}

```

Es un cálculo simple. Usamos @Depends para recalcular y revisualizar vat cada vez que el usuario modifique vatPercentage.

Solo nos queda totalAmount por añadir. Puedes ver su código en el listado 8.14.

Listado 5.14 La propiedad calculada totalAmount en CommercialDocument

```

@Stereotype("MONEY")
@Depends("baseAmount, vat") // Cuando baseAmount o vat cambian totalAmount se
public BigDecimal getTotalAmount() { // recalcula y revisualiza
    return getBaseAmount().add(getVat()); // baseAmount + vat
}

```

Una vez más un cálculo simple, y una vez más usamos @Depends.

Ahora que ya has escrito las propiedades para los importes de CommercialDocument, tienes que modificar la vista para que estas nuevas propiedades se muestren (listado 5.15).

Listado 5.15 Añadir propiedades de importes a la vista de CommercialDocument

```

@View(members=
    "year, number, date;" +
    "data {" +

```

```

        "customer;" +
        "details;" +
        "amounts [ " + // Los corchetes indican un grupo, visualizado dentro de un marco
            "vatPercentage, baseAmount, vat, totalAmount" +
        "];" +
        "remarks" +
    "}"
)
abstract public class CommercialDocument extends Identifiable {

```

Añadimos `vatPercentage`, `baseAmount`, `vat`, `totalAmount` entre `details` y `remarks`, pero entre unos corchetes con un nombre (`amounts`). Esto significa que estas propiedades se mostrarán dentro de un marco llamado “amount”. Esto es un grupo.

Ahora, puedes actualizar el esquema de la base de datos (por causa de `vatPercentage`) y probar tu aplicación. Debería funcionar casi como en la susodicha figura 5.3. “Casi” porque `vatPercentage` todavía no tiene un valor por defecto. Lo añadiremos en la siguiente sección.

5.1.4 Valor por defecto desde un archivo de propiedades

Es conveniente para el usuario tener el campo `vatPercentage` lleno por defecto con un valor adecuado. Puedes usar un calculador (`@DefaultValueCalculator`) que devuelva un valor fijo, en este caso cambiar el valor por defecto implica cambiar el código fuente. O puedes leer el valor por defecto de una base de datos (usando JPA desde tu calculador), en este caso cambiar el valor por defecto implica actualizar la base de datos.

Otra opción es tener estos valores de configuración en un archivo de propiedades, un archivo plano con pares clave=valor. En este caso cambiar el valor por defecto de `vatPercentage` es tan simple como editar un archivo plano con un editor de texto.

Implementemos la opción del archivo de propiedades. Crea un archivo llamado *invoicing.properties* en la carpeta *Invoicing/properties* con el contenido del listado 5.16.

Listado 5.16 El contenido del archivo *invoicing.properties*

```
defaultVatPercentage=21
```

Aunque puedes usar la clase `java.util.Properties` de Java para leer este archivo preferimos usar una clase propia para leer estas propiedades. Vamos a llamar a esta clase `InvoicingPreferences` y la pondremos en un nuevo paquete llamado `org.openxava.invoicing.util`. Tienes el código en el listado 5.17.

Listado 5.17 InvoicingPreferences para leer el archivo invoicing.properties

```

package org.openxava.invoicing.util; // En el paquete util

import java.io.*;
import java.math.*;
import java.util.*;

import org.apache.commons.logging.*;
import org.openxava.util.*;

public class InvoicingPreferences {

    private final static String
        FILE_PROPERTIES="invoicing.properties";
    private static Log log =
        LoggerFactory.getLog(InvoicingPreferences.class);
    private static Properties properties; // Almacenamos las propiedades aquí

    private static Properties getProperties() {
        if (properties == null) { // Usamos inicialización vaga
            PropertiesReader reader = // PropertiesReader es una clase de OpenXava
                new PropertiesReader(
                    InvoicingPreferences.class,
                    FILE_PROPERTIES);
            try {
                properties = reader.get();
            }
            catch (IOException ex) {
                log.error(
                    XavaResources.getString( ← // Para leer un mensaje i18n
                        "properties_file_error",
                        FILE_PROPERTIES),
                    ex);
                properties = new Properties();
            }
        }
        return properties;
    }

    public static BigDecimal getDefaultVatPercentage() { // El único método público
        return new BigDecimal(
            getProperties().getProperty("defaultVatPercentage"));
    }
}

```

Como puedes ver InvoicingPreferences es una clase con un método estático, getDefaultVatPercentage(). La ventaja de usar esta clase en lugar de leer directamente del archivo de propiedades es que si cambias la forma en que se obtienen las preferencias, por ejemplo leyendo de una base de datos o de un directorio LDAP, solo has de cambiar esta clase en toda tu aplicación.

Puedes usar esta clase desde el calculador por defecto para la propiedad vatPercentage. El listado 5.18 contiene el código del calculador.

Listado 5.18 Calculador para el valor por defecto del porcentaje de IVA

```

package org.openxava.invoicing.calculators; // En el paquete calculators

import org.openxava.calculators.*; // Para usar ICalculator
import org.openxava.invoicing.util.*; // Para usar InvoicingPreferences

public class VatPercentageCalculator implements ICalculator {

    public Object calculate() throws Exception {
        return InvoicingPreferences.getDefaultVatPercentage();
    }

}

```

Como ves, simplemente devuelve `defaultValuePercentage` de `InvoicingPreferences`. Ahora, ya puedes usar este calculador en la definición de la propiedad `vatPercentage` en `CommercialDocument`. Mira en el listado 5.19.

Listado 5.19 Calculador por defecto para `vatPercentage` de `CommercialDocument`

```

@DefaultValueCalculator(VatPercentageCalculator.class)
private BigDecimal vatPercentage;

```

Con este código cuando el usuario pulsa para crear una nueva factura, el campo `vatPercentage` se rellenará con 21, o cualquier otro valor que hayas puesto en *invoicing.properties*.

5.2 Métodos de retrollamada JPA

Otra forma práctica de añadir lógica de negocio a tu modelo es mediante los métodos de retrollamada JPA. Un método de retrollamada se llama en un momento específico del ciclo de vida de la entidad como objeto persistente. Es decir, puedes especificar cierta lógica a ejecutar al grabar, leer, borrar o modificar una entidad.

En esta sección veremos algunas aplicaciones prácticas de los métodos de retrollamada JPA.

5.2.1 Cálculo de valor por defecto multiusuario

Hasta ahora estamos calculando el número para `Invoice` y `Order` usando `@DefaultValueCalculator`. Éste calcula el valor por defecto en el momento que el usuario pulsa para crear una nueva `Invoice` o `Order`. Por tanto, si varios usuarios pulsaran en el botón “nuevo” al mismo tiempo todos ellos obtendrán el mismo número. Esto no es apto para aplicaciones multiusuario. La forma correcta de generar un número único es generándolo justo en el momento de grabar.

93 Lección 5: Lógica de negocio básica

Vamos a implementar la generación del número usando métodos de retrollamada JPA. JPA permite marcar cualquier método de tu clase para ser ejecutado en cualquier momento de su ciclo de vida. Indicaremos que justo antes de grabar un `CommercialDocument` calcule su número. De paso mejoraremos el cálculo para tener una numeración diferente para `Order` e `Invoice`.

Edita la entidad `CommercialDocument` y añade el método `calculateNumber()` que tienes en el listado 5.20.

Listado 5.20 Método `@PrePersist calculateNumber()` de `CommercialDocument`

```
@PrePersist // Ejecutado justo antes de grabar el objeto por primera vez
public void calculateNumber() throws Exception {
    Query query = XPersistence.getManager()
        .createQuery("select max(i.number) from " +
            getClass().getSimpleName() + // De esta forma es válido para Invoice y Order
            " i where i.year = :year");
    query.setParameter("year", year);
    Integer lastNumber = (Integer) query.getSingleResult();
    this.number = lastNumber == null ? 1 : lastNumber + 1;
}
```

El código del listado 5.20 es el mismo que el de `NextNumberForYearCalculator` pero usando `getClass().getSimpleName()` en lugar de “`CommercialDocument`”. El método `getSimpleName()` devuelve el nombre de la clase sin paquete, es decir, precisamente el nombre de entidad. Será “`Order`” para `Order` e “`Invoice`” para `Invoice`. Así podemos obtener una numeración diferente para `Order` e `Invoice`.

La especificación JPA establece que no puedes usar el API JPA dentro de un método de retrollamada. Por tanto, el método de arriba no es legal desde un punto de vista estricto. Pero, Hibernate (la implementación de JPA que OpenXava usa por defecto) te permite usar en `@PrePersist`. Y dado que usar JPA es la forma más fácil de hacer este cálculo, nosotros lo usamos.

Ahora borra la clase `NextNumberForYearCalculator` de tu proyecto, y modifica la propiedad `number` de `CommercialDocument` para que no la use (listado 5.21).

Listado 5.21 `number` de `CommercialDocument` sin `@DefaultValueCalculator`

```
@Column(length=6)
@DefaultValueCalculator(value=NextNumberForYearCalculator.class, // Quita esto
    properties=@PropertyValue(name="year"))
+
@ReadOnly ← // El usuario no puede modificar el valor
private int number;
```

Nota que, además de quitar `@DefaultValueCalculator`, hemos añadido la anotación `@ReadOnly`. Esto significa que el usuario no puede introducir ni modificar este número. Esta es la forma correcta de hacerlo ahora dado que el

número es generado al grabar el objeto, por lo que el valor que tecleara el usuario sería sobrescrito siempre.

Prueba ahora el módulo de Invoice u Order, verás como el número está vacío y no es editable, y cuando añades la primera línea de detalle, lo cual graba el documento contenedor, el número de documento se calcula y se actualiza en la interfaz de usuario.

5.2.2 Sincronizar propiedades persistentes y calculadas

La forma en que calculamos el IVA, el importe base y el importe total es natural y práctica. Usamos propiedades calculadas que calculan, usando Java puro, los valores cada vez que son llamadas. Esto está bien porque cada vez que el usuario añade, modifica o borra una línea de detalle en la interfaz de usuario, el IVA, el importe base y el importe total se recalculan con datos frescos instantáneamente.

Pero, las propiedades calculadas tienen algunos inconvenientes. Por ejemplo, si quieres hacer un proceso masivo o un informe de todas las facturas cuyo importe total esté entre ciertos rangos. En estos casos, si tienes una base de datos demasiado grande el proceso puede ser lentísimo, porque has de instanciar todas las facturas para calcular su importe total. Una solución para este problema es tener una propiedad persistente, por tanto una columna en la base de datos para el importe de la factura o pedido; así el rendimiento es bastante mayor.

En nuestro caso mantendremos nuestra actuales propiedades calculadas, pero vamos a añadir una nueva, llamada `amount`, que contendrá el mismo valor que `totalAmount`, pero `amount` será persistente con su correspondiente columna en la base de datos. Lo complicado aquí es mantener sincronizado el valor de la propiedad `amount`. Vamos a usar métodos de retrollamada JPA para conseguirlo.

El primer paso es añadir la propiedad `amount` a `CommercialDocument`. Nada más fácil, puedes verlo en el listado 5.22.

Listado 5.22 Propiedad persistente `amount` en `CommercialDocument`

```
@Stereotype("MONEY")
private BigDecimal amount;

public BigDecimal getAmount() {
    return amount;
}

public void setAmount(BigDecimal amount) {
    this.amount = amount;
}
```

Modifiquemos la entidad `Detail` para que cada vez que un detalle sea

95 Lección 5: Lógica de negocio básica

añadido, quitado o modificado la propiedad `amount` de su `CommercialDocument` contenedor sea recalculada correctamente. Añade los tres métodos de retrollamada del listado 5.23 a tu entidad `Detail`.

Listado 5.23 Métodos de retrollamada en `Detail` recalculan el importe del padre

```
@PrePersist // Al grabar el detalle por primera vez
private void onPersist() {
    getParent().getDetails().add(this); // Para tener la colección sincronizada
    getParent().recalculateAmount();
}

@PreUpdate // Cada vez que el detalle se modifica
private void onUpdate() {
    getParent().recalculateAmount();
}

@PreRemove // Al borrar el detalle
private void onRemove() {
    getParent().getDetails().remove(this); // Para tener la colección sincronizada
    getParent().recalculateAmount();
}
```

Llamamos al método `recalculateAmount()` del `CommercialDocument` padre cada vez que se añade, quita o modifica un detalle. Veamos este método en el listado 5.24.

Listado 5.24 `recalculateAmount()` sincroniza `totalAmount` y `amount`

```
public void recalculateAmount() {
    setAmount(getTotalAmount());
}
```

Como puedes ver movemos la propiedad calculada, `totalAmount`, a la persistente, `amount`.

Prueba el módulo `Invoice` u `Order` con este código y verás que cuando una línea de detalle es añadida, borrada o modificada la columna en la base de datos para `amount` se actualiza correctamente. Pero, si tratas de borrar la `Invoice` u `Order` obtendrás una `java.util.ConcurrentModificationException`. Esto es porque estamos usando *cascade ALL*, y cuando un `CommercialDocument` se borra sus detalles son borrados automáticamente, y en este caso el método de retrollamada `@PreRemove` (`onRemove()`) falla. Hemos de hacer un pequeño ajuste para evitar este desagradable efecto. El ajuste es simple: no ejecutar el método `onRemove()` de `Detail` cuando el `CommercialDocument` contenedor está siendo borrado. Para programar esto añade el código del listado 5.25 a tu clase `CommercialDocument`.

Listado 5.25 Código en `CommercialDocument` para indicar que se está borrando

```
@Transient // No se almacena en la tabla de la base de datos
private boolean removing = false; // Indica si JPA está borrando el documento ahora
```

```

boolean isRemoving() { // Acceso paquete, no es accesible desde fuera
    return removing;
}

@PreRemove // Cuando el documento va a ser borrado marcamos removing como true
private void markRemoving() {
    this.removing = true;
}

@PostRemove // Cuando el documento ha sido borrado marcamos removing como false
private void unmarkRemoving() {
    this.removing = false;
}

```

Aquí ves como hemos añadido una propiedad transitoria removing, que es *true* solo cuando el CommercialDocument está siendo borrado. Podemos usar esta propiedad desde Detail de la forma que ves en el listado 5.26.

Listado 5.26 Método onRemoved() de Detail refinado

```

@PreRemove
private void onRemove() {
    if (getParent().isRemoving()) return; // Añadimos esta línea para evitar excepciones
    getParent().getDetails().remove(this);
    getParent().recalculateAmount();
}

```

Es decir, si el contenedor está siendo borrado, no ejecutamos la lógica de onRemove(). Después de este pequeño ajuste, Invoice y Order tienen su propiedad amount siempre sincronizada, y lista para ser usada en un proceso masivo.

5.3 Lógica desde la base de datos (@Formula)

Idealmente escribirás toda tu lógica de negocio en Java, dentro de tus entidades. Sin embargo, hay ocasiones que esto no es lo más conveniente. Imagina que tienes una propiedad calculada en CommercialDocument, digamos estimatedProfit, como la del listado 5.27.

Listado 5.27 estimatedProfit como propiedad calculada en CommercialDocument

```

@Stereotype("MONEY")
public BigDecimal getEstimatedProfit() {
    return getAmount().multiply(new BigDecimal("0.10"));
}

```

Si necesitas realizar un proceso con todas las facturas con un estimatedProfit mayor de 1000, has de escribir algo parecido al listado 5.28.

Listado 5.28 Seleccionar objetos según una propiedad calculada

```

Query query = getManager()
    .createQuery("from Invoice"); // Sin condición en la consulta
for (Object o: query.getResultList()) { // Itera por todos los objetos
    Invoice i = (Invoice) o;
    if (i.getEstimatedProfit() // Pregunta a cada objeto
        .compareTo(new BigDecimal("1000")) > 0) {
        i.doSomething();
    }
}

```

No puedes usar una condición en la consulta para discriminar por `estimatedProfit`, porque `estimatedProfit` no está en la base de datos, solo está en el objeto Java, por tanto tienes que instanciar cada objeto para preguntar por su `estimatedProfit`. A veces esto es una buena opción, pero si tienes una cantidad inmensa de facturas, y solo unas cuantas tienen el `estimatedProfit` mayor de 1000, entonces el proceso será muy ineficiente. ¿Qué alternativa tenemos?

Nuestra alternativa es usar la anotación `@Formula`. `@Formula` es una extensión de Hibernate al JPA estándar, que te permite mapear tu propiedad contra un estamento SQL. Puedes definir `estimatedProfit` con `@Formula` como muestra el listado 5.29.

Listado 5.29 estimatedProfit con @Formula en CommercialDocument

```

@org.hibernate.annotations.Formula("AMOUNT * 0.10") // El cálculo usando SQL
@Stereotype("MONEY")
private BigDecimal estimatedProfit; // Un campo, como con las propiedades persistentes

public BigDecimal getEstimatedProfit() { // Sólo el getter es necesario
    return estimatedProfit;
}

```

Esto indica que cuando un `CommercialDocument` se lea de la base de datos, el campo `estimatedProfit` se rellenará con el cálculo de `@Formula`, un cálculo que por cierto hace la base de datos. Lo más útil de las propiedades `@Formula` es que puedes usarlas en las condiciones, por tanto puedes reescribir el anterior proceso como muestra el listado 5.30.

Listado 5.30 Seleccionar objetos según una propiedad @Formula

```

Query query = getManager()
    .createQuery("from Invoice i where " +
        "i.estimatedProfit > :estimatedProfit"); // Podemos usar una condición
query.setParameter("estimatedProfit", new BigDecimal(1000));
for (Object o: query.getResultList()) { // Iteramos solo por los objetos seleccionados
    Invoice i = (Invoice) o;
    i.doSomething();
}

```

De esta forma pones el peso del cálculo de `estimatedProfit` y la selección

de los registros en el servidor de base de datos, y no el servidor Java.

Este hecho también tiene efecto en modo lista, porque el usuario no puede filtrar ni ordenar por propiedades calculadas, pero sí por propiedades con @Formula (figura 5.4)

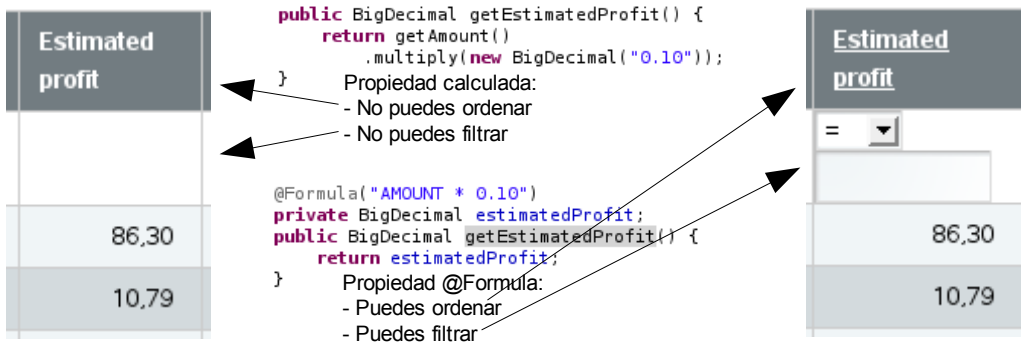


Figura 5.4 Modo lista para propiedades calculadas y propiedades @Formula

@Formula es una buena opción para mejorar el rendimiento en algunos casos. De todas formas, generalmente es mejor usar propiedades calculadas y escribir así tu lógica en Java. La ventaja de las propiedades calculadas sobre @Formula es que tu código no es dependiente de la base de datos. Además, con las propiedades calculadas puedes reejecutar el cálculo sin tener que leer el objeto de la base de datos, por tanto puedes usar @Depends.

5.4 Pruebas JUnit

Antes de ir a la siguiente lección, vamos a escribir el código JUnit para éste. Recuerda, el código no está terminado si no tiene pruebas JUnit. Puedes escribir las pruebas antes, durante o después del código principal. Pero siempre has de escribirlas.

El código de prueba mostrado aquí no es solo para darte un buen ejemplo, sino también para enseñarte maneras de probar diferentes casos en tu aplicación OpenXava.

5.4.1 Modificar la prueba existente

Crear una nueva prueba para cada nuevo caso parece una buena idea desde un punto de vista estructural, pero en la mayoría de los casos no es práctico, porque de esa forma tu código de prueba crecerá muy rápido, y con el tiempo, ejecutar todas las pruebas supondrá muchísimo tiempo.

El enfoque más pragmático es modificar el código de prueba existente para

99 Lección 5: Lógica de negocio básica

cubrir todos los nuevos casos que hemos desarrollado. Hagámoslo de esta forma.

En nuestro caso, todo el código de esta lección aplica a `CommercialDocument`, por tanto vamos a modificar el método `testCreate()` de `CommercialDocumentTest` para ajustarlo a la nueva funcionalidad. Dejamos el método `testCreate()` tal como muestra el listado 5.31.

Listado 5.31 El método `testCreate()` de `CommercialDocumentTest` modificado

```
public void testCreate() throws Exception {
    calculateNumber(); // Añadido para calcular primero el siguiente número de documento
    verifyDefaultValues();
    chooseCustomer();
    addDetails();
    setOtherProperties();
    save();
    verifyAmountAndEstimatedProfit(); // Prueba el método de retrollamada y @Formula
    verifyCreated();
    remove();
}
```

Como ves, añadimos una nueva línea al principio para calcular el siguiente número de documento, y una llamada al nuevo método `verifyAmountAndEstimatedProfit()`.

Ahora nos conviene más calcular el siguiente número de documento al principio para usarlo en el resto de la prueba. Para hacer esto, cambia el viejo método `getNumber()` por los dos métodos mostrados en el listado 5.32.

Listado 5.32 Un método para calcular el siguiente número y otro para leerlo

```
private void calculateNumber() {
    Query query = getManager().
        createQuery(
            "select max(i.number) from " +
            model + // Cambiamos CommercialDocument por una variable
            " i where i.year = :year");
    query.setParameter("year", Dates.getYear(new Date()));
    Integer lastNumber = (Integer) query.getSingleResult();
    if (lastNumber == null) lastNumber = 0;
    number = Integer.toString(lastNumber + 1);
}

private String getNumber() {
    return number;
}
```

Anteriormente, teníamos solo `getNumber()` que calculaba y devolvía el número, ahora tenemos un método para calcular (`calculateNumber()`), y otro para devolver el resultado (`getNumber()`). Puedes notar que la lógica del cálculo tiene un pequeño cambio, en vez de usar “`CommercialDocument`” como fuente de la consulta usamos `model`, una variable. Esto es así porque ahora la numeración para facturas y pedidos está separada. Llenamos esta variable, un

campo de la clase de prueba, en el constructor, tal como muestra el listado 5.33.

Listado 5.33 El constructor de la prueba almacena el nombre del modelo

```
private String model; // Nombre del modelo para la condición. Puede ser "Invoice" u "Order"

public CommercialDocumentTest(String testName, String moduleName) {
    super(testName, "Invoicing", moduleName);
    this.model = moduleName; // El nombre del módulo coincide con el del modelo
}
```

En este caso el nombre de módulo, “Invoice” u “Order”, coincide con el nombre de modelo, “Invoice” u “Order”, así que la forma más fácil de obtener el nombre de modelo es desde el nombre de módulo.

Veamos el código que prueba la nueva funcionalidad.

5.4.2 Verificar valores por defecto y propiedades calculadas

En esta lección hemos hecho algunas modificaciones en los valores por defecto. Primero, el valor por defecto para number ya no se calcula mediante un @DefaultValueCalculator en su lugar usamos un método de retrollamada JPA. Segundo, tenemos una nueva propiedad, vatPercentage, cuyo valor inicial se calcula leyendo de un archivo de propiedades. Para probar estos casos hemos de modificar el método verifyDefaultValues() como ves en el listado 5.34.

Listado 5.34 El método verifyDefaultValues() adaptado

```
private void verifyDefaultValues() throws Exception {
    execute("CRUD.new");
    assertValue("year", getCurrentYear());
    assertValue("number", getNumber()); // Ahora el número no tiene valor inicial
    assertValue("number", ""); // al crear un documento nuevo (sección 5.2.1)
    assertValue("date", getCurrentDate());
    assertValue("vatPercentage", "21"); // Valor de archivo de propiedades (sección 5.1.4)
}
```

Comprobamos el cálculo del valor por defecto de vatPercentage y verificamos que number no tiene valor inicial, porque ahora number no se calcula hasta el momento de grabar el documento (sección 5.2.1). Cuando el documento (factura o pedido) se grabe verificaremos que number se calcula. El documento se graba justo cuando la primera línea de detalle se añade. También cuando la línea se añade podemos verificar el cálculo de amount de Detail (la propiedad calculada simple, sección 5.1.1), el valor por defecto para pricePerUnit (@DefaultValueCalculator, sección 5.1.2) y las propiedades de importes del documento (propiedades calculadas que dependen de una colección, sección 5.1.3). Probamos todo esto haciendo unas ligeras modificaciones en el ya existente método addDetails() (listado 5.35).

Listado 5.35 addDetails() prueba propiedades calculadas y valores por defecto

```

private void addDetails() throws Exception {
    // Añadir una línea de detalle
    assertCollectionRowCount("details", 0);
    execute("Collection.new",
        "viewObject=xava_view_section0_details");
    setValue("product.number", "1");
    assertValue("product.description",
        "Peopleware: Productive Projects and Teams");
    assertValue("pricePerUnit", // @DefaultValueCalculator, sección 5.1.2
        "31.00");
    setValue("quantity", "2");
    assertValue("amount", "62.00"); // Propiedad calculada, sección 5.1.1
    execute("Collection.save");
    assertNoErrors();
    assertCollectionRowCount("details", 1);

    // Al grabar el primer detalle se graba el documento,
    // entonces verificamos que el número se calcula
    assertValue("number", getNumber()); // Valor por defecto multiusuario, sección 5.2.1

    // Verificar propiedades calculadas del documento
    assertValue("baseAmount", "62.00"); // Propiedades calculadas
    assertValue("vat", "11.16"); // que dependen de una colección,
    assertValue("totalAmount", "73.16"); // sección 5.1.3

    // Añadir otro detalle
    execute("Collection.new",
        "viewObject=xava_view_section0_details");
    setValue("product.number", "2");
    assertValue("product.description",
        "Arco iris de lágrimas");
    assertValue("pricePerUnit", // @DefaultValueCalculator, sección 5.1.2
        "15.00");
    setValue("pricePerUnit", "10.00"); // Modificar el valor por defecto
    setValue("quantity", "1");
    assertValue("amount", "10.00"); // Propiedad calculada, sección 5.1.1
    execute("Collection.save");
    assertNoErrors();
    assertCollectionRowCount("details", 2);

    // Verificar propiedades calculadas del documento
    assertValue("baseAmount", "72.00"); // Propiedades calculadas
    assertValue("vat", "12.96"); // que dependen de una colección,
    assertValue("totalAmount", "84.96"); // sección 5.1.3
}

```

Como ves, con estas modificaciones sencillas probamos la mayoría de nuestro nuevo código. Nos quedan sólo las propiedades `amount` y `estimatedProfit`. Las cuales probaremos en la siguiente sección.

5.4.3 Sincronización entre propiedad persistente y calculada / @Formula

En la sección 5.2.2 usamos un método de retrollamada de JPA en

CommercialDocument para tener una propiedad persistente, amount, sincronizada con una calculada, totalAmount. La propiedad amount solo se muestra en modo lista.

En la sección 5.3 hemos creado una propiedad que usa @Formula, estimatedProfit. Esta propiedad se muestra solo en modo lista.

Obviamente, la forma más simple de probarlo es yendo a modo lista y verificando que los valores para estas dos propiedades son los esperados. En testCreate() llamamos a verifyAmountAndEstimatedProfit(). Veamos su código en el listado 5.36.

Listado 5.36 El nuevo método verifyAmountAndEstimatedProfit()

```
private void verifyAmountAndEstimatedProfit() throws Exception {
    execute("Mode.list"); // Cambia a modo lista
    setConditionValues(new String [] { // Filtra para ver en la lista solamente
        getFullYear(), getNumber() // el documento que acabamos de crear
    });
    execute("List.filter"); // Hace el filtro
    assertValueInList(0, 0, getFullYear()); // Verifica que el filtro
    assertValueInList(0, 1, getNumber()); // ha funcionado
    assertValueInList(0, "amount", "84.96"); // Confirma el importe
    assertValueInList(0, "estimatedProfit", "8.50"); // Confirma el beneficio estimado
    execute("Mode.detailAndFirst"); // Va a modo detalle
}
```

Dado que ahora vamos a modo lista y después volvemos a detalles, hemos de hacer una pequeña modificación en el método verifyCreated(), que es ejecutado justo después de verifyAmountAndEstimatedProfit(). Veamos la modificación en el listado 5.37.

Listado 5.37 verifyCreated() ahora no busca el documento recién creado

```
private void verifyCreated() throws Exception {
    setValue("year", getFullYear()); // Borramos estas líneas
    setValue("number", getNumber()); // para buscar el documento
    execute("CRUD.search"); // porque ya lo hemos buscado desde el modo lista

    // El resto de la prueba...
    ...
}
```

Quitamos estas líneas porque ahora no es necesario buscar el documento recién creado. Ahora en el método verifyAmountAndEstimatedProfit() vamos a modo lista y escogemos el documento, por tanto ya estamos editando el documento.

¡Enhorabuena! Ahora tus pruebas ya están sincronizadas con tu código. Es un buen momento para ejecutar todas las pruebas de tu aplicación.

5.5 Resumen

En esta lección has aprendido algunas formas comunes de añadir lógica de negocio a tus entidades. No hay duda sobre la utilidad de las propiedades calculadas, los métodos de retrollamada o `@Formula`. Sin embargo, todavía tenemos muchas otras formas de añadir lógica a tu aplicación OpenXava, que vamos a aprender a usar.

En futuras lecciones verás como añadir validación, modificar el funcionamiento estándar del módulo y añadir tu propia lógica de negocio, entre otras formas de añadir lógica personalizada a tu aplicación.